

Efficient Network Measurements through Approximated Windows

Ran Ben Basat Gil Einziger Roy Friedman
 Department of Computer Science
 Technion, Haifa 32000, Israel
 {sran, gilga, roy}@cs.technion.ac.il

Abstract

Many networking applications require timely access to recent network measurements, which can be captured using a sliding window model. Maintaining such measurements is a challenging task due to the fast line speed and scarcity of fast memory in routers. In this work, we study the efficiency factor that can be gained by approximating the window size. That is, we allow the algorithm to dynamically adjust the window size between W and $W(1 + \tau)$ where τ is a small positive parameter. For example, consider the *basic summing* problem of computing the sum of the last W elements in a stream whose items are integers in $\{0, 1, \dots, R\}$, where $R = \text{poly}(W)$. While it is known that $\Omega(W \log R)$ bits are needed in the exact window model, we show that approximate windows allow an exponential space reduction for constant τ .

Specifically, we present a lower bound of $\Omega(\tau^{-1} \log(RW\tau))$ bits for the basic summing problem. Further, an $(1 + \epsilon)$ multiplicative approximation of this problem requires $\Omega(\log(W/\epsilon) + \log \log R)$ bits for constant τ . Additionally, for $RW\epsilon$ additive approximations, we show an $\Omega(\tau^{-1} \log[1 + \tau/\epsilon] + \log(W/\epsilon))$ lower bound¹. For all three settings, we provide memory optimal algorithms that operate in constant time. Finally, we demonstrate the generality of the approximated window model by applying it to counting the number of distinct flows in a sliding window over a network stream. We present an algorithm that solves this problem while requiring asymptotically less space than previous sliding window methods when $\tau = O(1)$.

1 Introduction

Network algorithms in diverse areas such as traffic engineering, load balancing and quality of service [1, 8, 23, 27, 35] rely on timely link measurements. In such applications recent data is often more relevant than older one, motivating the notions of *aging* and *sliding window* [5, 10, 17, 28, 30]. For example, a sudden decrease in the average packet size on a link may indicate a SYN attack [29]. Additionally, a load balancer may benefit from knowing the current utilization of a link to avoid congestion.

While conceptually simple, conveying the necessary information to network algorithms is a difficult challenge due to the limitations of current memory technology. Specifically, DRAM memory is abundant but too slow to cope with the line rate while SRAM memory is fast enough but has a limited capacity [9, 14, 33]. On-line decisions are therefore realized through space efficient data structures [6, 7, 18, 19, 26, 32, 34] that store measurement statistics in a concise manner. For example, [18, 32] utilize probabilistic counters that only require $O(\log \log N)$ bits to

¹ We also provide an optimal bound and algorithm for the $\tau < \epsilon$ case.

approximately represent numbers up to N . Others conserve space using variable sized counter encoding [19, 26] and monitoring only the frequent elements [5].

BASIC-COUNTING is one of the most basic textbook examples of such approximated sliding window stream processing problems [15]. In this problem, one is required to keep track of the number of 1's in the last W elements of a stream of binary bits. In their seminal work, [15] presents a $(1 + \epsilon)$ -multiplicative approximation algorithm for this problem using $O(\frac{1}{\epsilon} \log^2 W \epsilon)$ bits, as well as a matching lower bound. Their solution works with amortized $O(1)$ time, but its worst case time complexity is $O(\log W)$. Additionally, the work of [4] provides a $RW\epsilon$ -additive approximation to the problem using $O(\frac{1}{\epsilon} + \log W \epsilon)$ bits as well as a matching lower bound.

A more practical variant is BASIC-SUMMING in which the goal is to maintain the sum of the last W elements. When all elements are non-negative integers in the range $[R+1] = \{0, 1, \dots, R\}$, the work in [15] naturally extends to provide a $(1 + \epsilon)$ -multiplicative approximation of this problem using $O(\frac{1}{\epsilon} \cdot (\log^2 W + \log R \cdot (\log W + \log \log R)))$ bits. The amortized time complexity becomes $O(\frac{\log R}{\log W})$ and the worst case is $O(\log W + \log R)$. In contrast, we showed an $RW\epsilon$ -additive approximation with $\Theta(\frac{1}{\epsilon} + \log W \epsilon)$ bits [4].

Sliding window counters (approximated or accurate) require asymptotically more space than plain stream counters. Such window counters are prohibitively large for networking devices which already optimize the space consumption of plain counters.

This paper explores the concept of *slack*, or approximated sliding window, to bridge this gap. Figure 1 illustrates a “window” in this model. Here, each query may select a τ -slacky window whose size is between W (the green elements) and $W(1 + \tau)$ (the green plus yellow elements). The goal is to compute the sum with respect to this chosen window.

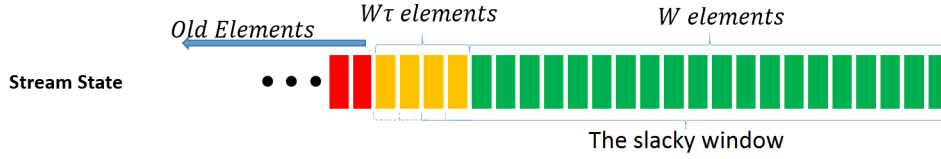


Figure 1: We need to answer each query with respect to a τ -slacky window that must include the last W items, but may or may not consider a suffix of the previous $W\tau$ elements.

Slacky windows were also considered in previous works [15, 30] and we call the problem of maintaining the sum over a slacky window SLACKY SUMMING. Datar et al. [15] showed that while computing a $(1 + \epsilon)$ -multiplicative approximation for BASIC-COUNTING requires $\Omega(\epsilon^{-1} \log^2 W \epsilon)$ memory bits. When $\tau = 1$, they can compute such an approximation using only $O(\epsilon^{-1} \log(W \epsilon) \log \log W)$ bits. For the BASIC-SUMMING problem, a constant slack reduces the memory requirement from $\Omega(\epsilon^{-1} \log W \log(RW))$ to $O(\epsilon^{-1} \log(RW) \log \log(RW))$. For τ -slacky windows, their space consumption is $O(\epsilon^{-1} \log(W \epsilon) (\log \log W + \log \tau^{-1}))$ and $O(\epsilon^{-1} \log(RW) (\log \log(RW) + \log \tau^{-1}))$ for providing a $(1 + \epsilon)$ -multiplicative approximation for BASIC-COUNTING and BASIC-SUMMING, respectively.

Our Contributions First, we show lower bounds on the space required by deterministic algorithms to solve SLACKY SUMMING. For exact SLACKY SUMMING, we show a lower bound of $\Omega(\tau^{-1} \log(RW\tau))$ bits. For a $(1 + \epsilon)$ multiplicative approximation we show a $\Omega(\log(W/\epsilon) + \log \log R + \tau^{-1} (\log \epsilon^{-1} + \log \log \log(RW\tau)))$ bits lower bound. If we allow an $RW\epsilon$ additive error, we show that $\Omega(\tau^{-1} \log \lfloor 1 + \tau/\epsilon \rfloor + \log(W/\epsilon))$ bits are required.

Next, we introduce algorithms for the SLACKY SUMMING problem, which asymptotically reduce the required memory compared to the sliding window model. For the exact and additive error versions of the problem, we provide memory optimal algorithms. In the multiplicative

error setting, we provide an $O(\tau^{-1} (\log \epsilon^{-1} + \log \log (RW\tau)) + \log(RW))$ space algorithm; this is asymptotically optimal when τ is constant and $R = \text{poly}(W)$, and asymptotically improves [15] for $\tau^{-1} = \omega(\epsilon^{-1} \log(RW))$. All our algorithms are deterministic and operate in worst case constant time.

To exemplify our results, consider monitoring the average bandwidth (in bytes per second) passed through a router in a 24 hours window, i.e., $W \triangleq 86400$. Assuming we use 100GbE fiber transceiver, our stream values are bounded by $R \approx 2^{34}$ bytes. If we are willing to withstand an error of $\epsilon = 2^{-20}$ (i.e., about 16KBps), the work of [4] provides an additive approximation over the sliding window but requires about 120KB. In contrast, using a 10 minutes slack ($\tau \triangleq \frac{1}{144}$), our algorithm for **exact** SLACKY SUMMING requires only 800 bytes, 99% less than approximate summing over exact sliding window. For the same slack size, the algorithm of [15] requires more space than our **exact** algorithm even for a large 3% error. Further, if we also allow the same additive error ($\epsilon = 2^{-20}$), we provide an algorithm that requires only 240 bytes - a reduction of more than 99.8% !

Our final contribution relates to applying the slacky window approach to the problem of *counting distinct* items. Here, we show that a constant τ slack asymptotically reduces the space requirement compared to the state of the art sliding window algorithms [10, 31].

2 Preliminaries

For $\ell \in \mathbb{N}$, we denote $[\ell] \triangleq \{0, 1, \dots, \ell\}$. We Consider a stream of data elements $S = x_1, x_2, \dots, x_t$, where at each step a new element $x_i \in [R]$ is added to S . A W -sized window contains only the last W elements: $x_{t-W+1} \dots x_t$. We say that \mathcal{F} is a τ -slacky W -sized window if there exists $c \in [W\tau - 1]$ such that $\mathcal{F} = x_{t-(W+c)+1} \dots x_t$. For simplicity, we assume that τ^{-1} and $W\tau$ are integers. Unless explicitly specified, the base of all logs is 2.

Algorithms for the SLACKY SUMMING problem are required to support two operations:

1. **UPDATE**(x_t) Process a new element $x_t \in [R]$.
2. **OUTPUT** () Return a pair $\langle \hat{S}, c \rangle$ such that $c \in \mathbb{N}$ is the slack size and \hat{S} is an estimation of the last $W + c$ elements sum, i.e., $S \triangleq \sum_{k=t-(W+c)+1}^t x_k$.

We consider three types of algorithms for SLACKY SUMMING:

1. **Exact algorithms:** an algorithm \mathbb{A} solves the (W, τ) -EXACT SUMMING if its **OUTPUT** returns $\langle \hat{S}, c \rangle$ that satisfies $0 \leq c < W\tau$ and $\hat{S} = S$.
2. **Additive algorithms:** we say that \mathbb{A} solves the (W, τ, ϵ) -ADDITIVE SUMMING if its **OUTPUT** command returns $\langle \hat{S}, c \rangle$ that satisfies $0 \leq c < W\tau$ and $|S - \hat{S}| < RW\epsilon$.
3. **Multiplicative algorithms:** we say that \mathbb{A} solves the (W, τ, ϵ) -MULTIPLICATIVE SUMMING if its **OUTPUT** returns $\langle \hat{S}, c \rangle$ that satisfies $0 \leq c < W\tau$ and $\frac{S}{1+\epsilon} < \hat{S} \leq S$ if $S > 0$, and $\hat{S} = 0$ otherwise.

3 Lower Bounds

In this section, we analyze the space required for solving the SLACKY SUMMING problems.

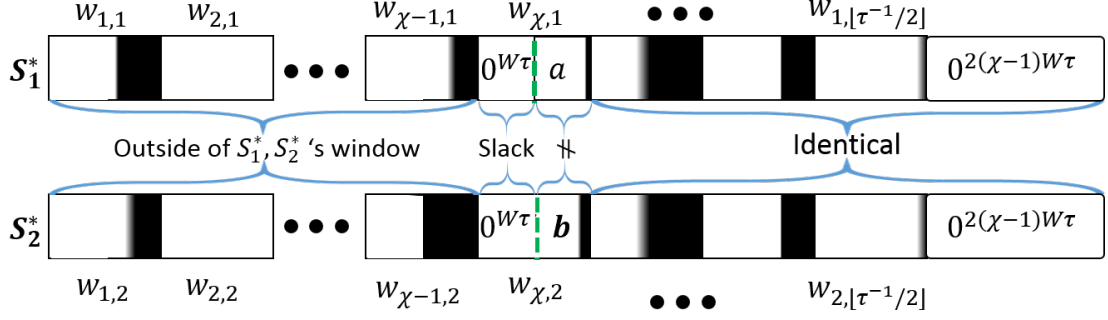


Figure 2: An illustration of the $\lceil \tau^{-1}/2 \rceil \log(RW\tau + 1)$ lower bound setting. If we assume that after seeing $w_{1,1} \cdot w_{2,1} \cdots w_{\lceil \tau^{-1}/2 \rceil,1}$ we reach the same configuration as after processing $w_{1,2} \cdot w_{2,2} \cdots w_{\lceil \tau^{-1}/2 \rceil,2}$, we will provide the wrong answer for at least one of S_1^*, S_2^* .

3.1 (W, τ) -EXACT SUMMING

We start by proving lower bounds on the memory required for exact SLACKY SUMMING.

Lemma 1. *Any deterministic algorithm \mathbb{A} that solves the (W, τ) -EXACT SUMMING problem must use at least $\lceil \log(RW(W+1)/2+1) \rceil \geq \lfloor \log(RW^2) \rfloor$ bits.*

We now use the lemma, whose proof is deferred to Appendix A, to show the following lower bound on exact (W, τ) -EXACT SUMMING algorithms:

Theorem 1. *Any deterministic algorithm \mathbb{A} that solves the (W, τ) -EXACT SUMMING problem must use at least $\max \{ \lfloor \log(RW^2) \rfloor, \lceil \lceil \tau^{-1}/2 \rceil \log(RW\tau + 1) \rceil \}$ bits.*

Proof. Lemma 1 shows a $\lfloor \log(RW^2) \rfloor$ bound. We proceed with showing a lower bound $\lceil \lceil \tau^{-1}/2 \rceil \log(RW\tau + 1) \rceil$ bits. Consider the following languages:

$$L_{E_2} \triangleq \{ 0^{W\tau+i} \sigma R^{W\tau-i-1} \mid i \in [W\tau - 1], \sigma \in [R] \}, \overline{L}_{E_2} \triangleq \{ w_1 \cdot w_2 \cdots w_{\lceil \tau^{-1}/2 \rceil} \mid \forall i : w_i \in L_{E_2} \}.$$

Notice that $|\overline{L}| = (RW\tau + 1)^{\lceil \tau^{-1}/2 \rceil}$. Our goal is to show that each input in \overline{L} must be mapped into a distinct memory configuration. Let $S_1 \triangleq w_{1,1} \cdot w_{2,1} \cdots w_{\lceil \tau^{-1}/2 \rceil,1}$, $S_2 \triangleq w_{1,2} \cdot w_{2,2} \cdots w_{\lceil \tau^{-1}/2 \rceil,2}$ be two distinct inputs in \overline{L} such that $\forall i : w_{i,1}, w_{i,2} \in L_{E_2}$. Denote $\chi \triangleq \max \{ i \in \lceil \tau^{-1}/2 \rceil \mid w_{i,1} \neq w_{i,2} \}$ – the last place in which S^1 differs from S^2 . Next, consider the sequences $S_1^* = S_1 \cdot 0^{2W(\chi-1)\tau}$ and $S_2^* = S_2 \cdot 0^{2W(\chi-1)\tau}$. Notice that the last W elements window for S_1^*, S_2^* is $w_{\chi,1} \cdots w_{\lceil \tau^{-1}/2 \rceil,1} \cdot 0^{2W(\chi-1)\tau}$ and $w_{\chi,2} \cdots w_{\lceil \tau^{-1}/2 \rceil,2} \cdot 0^{2W(\chi-1)\tau}$ respectively, and that the preceding $W\tau$ elements are all zeros. An illustration of the setting appears in Figure 2. By our choice of χ , we have that the sum of the last W elements of S_1^* and S_2^* is different, and since the slack is all zeros, no answer would be correct on both. Finally, we note that this implies that S^1, S^2 also had to reach different configurations, as otherwise \mathbb{A} would have to reach the same configuration after processing the additional $2W(\chi-1)\tau$ zeros. \square

3.2 (W, τ, ϵ) -ADDITIVE SUMMING

Next, we show a lower bound for additive approximations of SLACKY SUMMING. Due to lack of space, the proof is deferred to Appendix B.

Theorem 2. *For $\epsilon < 1/4$, any deterministic algorithm \mathbb{A} that solves the (W, τ, ϵ) -ADDITIVE SUMMING problem requires $\max \{ \log(W/\epsilon) - O(1), \lceil \lceil \tau^{-1}/2 \rceil \log \lfloor \tau/2\epsilon + 1 \rfloor \}$ bits.*

3.3 (W, τ, ϵ) -MULTIPLICATIVE SUMMING

In this section, our goal is to show lower bounds for multiplicative approximations of SLACKY SUMMING.

Lemma 2. *Consider the integer set $I_M \triangleq \{a_n \mid a_n \leq R \lfloor W/2 \rfloor\}$, where the integers $\{a_i\}$ are taken from the following sequence: $a_1 = 1, \forall n > 1 : a_n = \lceil (1 + \epsilon)a_{n-1} \rceil$. The cardinality of I_M satisfies $|I_M| \geq \ln(2) \epsilon^{-1} \log(RW\epsilon/2) - O(1)$.*

Proof. We first show an upper bound on $a_n \forall n \in \mathbb{N} : a_n \leq \epsilon^{-1} \cdot ((1 + \epsilon)^n - 1)$.

- **Basis:** for $n = 1$, we have $a_1 = 1 = \epsilon^{-1} \cdot ((1 + \epsilon) - 1)$.
- **Hypothesis:** $a_{n-1} \leq \epsilon^{-1} \cdot ((1 + \epsilon)^{n-1} - 1)$.
- **Step:** For $n > 1$, we bound a_n as follows:

$$\begin{aligned} a_n = \lceil (1 + \epsilon)a_{n-1} \rceil &\leq \lceil (1 + \epsilon)\epsilon^{-1} \cdot ((1 + \epsilon)^{n-1} - 1) \rceil \\ &= \lceil \epsilon^{-1}(1 + \epsilon)^n - \epsilon^{-1} - 1 \rceil < \epsilon^{-1} \cdot ((1 + \epsilon)^n - 1). \end{aligned}$$

Next, notice that this implies that $|I_M| \geq \arg \max \{n \mid \epsilon^{-1}(1 + \epsilon)^n \leq R \lfloor W/2 \rfloor\}$. Finally, we get a lower bound of $n = \lfloor \log_{1+\epsilon}(R \lfloor W/2 \rfloor \epsilon) \rfloor = \frac{\ln(RW\epsilon/2)}{\ln(1+\epsilon)} - O(1) < \ln(2) \epsilon^{-1} \log(RW\epsilon/2) - O(1)$, where the last inequality follows from the Taylor expansion of $\ln(1 + \epsilon)$. \square

We now use Lemma 2 to obtain a lower bound for (W, τ, ϵ) -MULTIPLICATIVE SUMMING.

Lemma 3. *For $\epsilon < 1/4$, any deterministic algorithm \mathbb{A} for the (W, τ, ϵ) -MULTIPLICATIVE SUMMING problem requires at least $\log(W/\epsilon) + \log \log(RW\epsilon) - O(1)$ memory bits.*

Proof. We show a language $\overline{L_M}$ for which every two words must reach a unique memory configuration, thus implying a $\lceil \log |\overline{L_M}| \rceil$ bits lower bound. We denote by $\text{rep}(x) \triangleq (x \bmod R) \cdot R^{\lfloor x/R \rfloor}$ a sequence in $\{\sigma R^* \mid \sigma \in [R]\}$ that has a sum of x . We define $\overline{L_M}$ as follows:

$$\overline{L_M} \triangleq \{0^{W+W\tau} \text{rep}(x) 0^j \mid j \in [\lfloor W/2 \rfloor], x \in I_M\}.$$

Notice that $|\overline{L_M}| = |I_M| \cdot (\lfloor W/2 \rfloor + 1)$ and according to Lemma 2 we have

$$\log |\overline{L_M}| \geq \log(\ln(2) \epsilon^{-1} \log(RW\epsilon/2) - O(1)) + \log W - 1 = \log(W/\epsilon) + \log \log(RW\epsilon) - O(1).$$

Consider two words $w_1 = 0^{W+W\tau} \text{rep}(x_1) 0^{j_1}$ and $w_2 = 0^{W+W\tau} \text{rep}(x_2) 0^{j_2}$ in $\overline{L_M}$, such that $x_1, x_2 \in I_M$. Notice that every two distinct numbers $z < q \in I_M$ satisfy $z \leq q/(1 + \epsilon)$. Since w_1, w_2 are both preceded with a sequence of $W\tau$ zeros, no \hat{S} answer correctly satisfies the requirements for both w_1, w_2 . Thus, if $x_1 \neq x_2$, \mathbb{A} must reach a different configuration after processing w_1 than it reaches when reading w_2 . Next, assume that $x_1 = x_2$, and that without loss of generality $j_1 < j_2$ (i.e., we have $w_2 = w_1 \cdot 0^{j_2-j_1}$). If the algorithm reached some configuration \mathbf{c} after reading w_1 and then returned to it after reading the additional $j_2 - j_1$ zeros of w_2 , it must get back to \mathbf{c} when reading $w_1 \cdot 0^{W(1+\tau)(j_2-j_1)}$. Notice that the sum of w_1 is x_1 and the current window sums to zero. Thus, no single \hat{S} value would satisfy both sequences. Hence, \mathbb{A} must reach a different configuration than \mathbf{c} when seeing w_2 . \square

To extend our multiplicative lower bound, we use the following fact:

Fact 1. *For any $x \neq 1, y \in \mathbb{R}$, the sequence $\{c_i\}_{i=1}^\infty$, defined as $c_n \triangleq \begin{cases} 1 & n = 1 \\ x \cdot c_{n-1} + y & \text{Otherwise} \end{cases}$ can be represented using closed form as $c_n = x^{n-1} + y \cdot \frac{x^n - 1}{x - 1}$.*

Next, let $k \in \mathbb{N}$ and $\psi, \epsilon \in \mathbb{R}$, such that $\psi \geq 2$, $\epsilon > 0$, $k \geq 1$; consider the integer sequence $\{a_{i,k}\}_{i=1}^{\infty}$, defined as

$$a_{n,k} \triangleq \begin{cases} 1 & n = 1 \\ \left\lceil (1 + \epsilon) \left(a_{n-1,1} + \sum_{i=1}^{k-1} \psi^i \right) \right\rceil & \text{Otherwise.} \end{cases}$$

Using the fact above, we show the following lemma:

Lemma 4. *For every integer $n \geq 1$ we have $a_{n,k} \leq 4\epsilon^{-1}(1 + \epsilon)^{n+1}\psi^{k-1}$.*

Proof. To apply Fact 1, we define an upper bounding sequence $\{b_{i,k}\}_{i=1}^{\infty}$ as follows:

$$b_{n,k} \triangleq \begin{cases} 1 & n = 1 \\ (1 + \epsilon) \left(b_{n-1,1} + \sum_{i=1}^{k-1} \psi^i \right) + 1 & \text{Otherwise.} \end{cases}$$

Thus, we can rewrite the n 'th element of the sequence as:

$$b_{n,k} = (1 + \epsilon)^{n-1} + \frac{(1 + \epsilon)^n - 1}{(1 + \epsilon) - 1} \left((1 + \epsilon) \sum_{i=1}^{k-1} \psi^i + 1 \right).$$

We can now use this representation to derive an upper bound of $b_{n,k}$:

$$\begin{aligned} b_{n,k} &= (1 + \epsilon)^{n-1} + \left((1 + \epsilon) \sum_{i=1}^{k-1} \psi^i + 1 \right) \frac{(1 + \epsilon)^n - 1}{(1 + \epsilon) - 1} \\ &\leq (1 + \epsilon)^{n-1} + ((1 + \epsilon)2\psi^{k-1}) \frac{(1 + \epsilon)^n - 1}{\epsilon} \leq 4\epsilon^{-1}(1 + \epsilon)^{n+1}\psi^{k-1}. \end{aligned}$$

Finally, since $a_{n,k} \leq b_{n,k}$ for any n, k , we conclude that $a_{n,k} \leq 4\epsilon^{-1}(1 + \epsilon)^{n+1}\psi^{k-1}$. \square

We now define the integer set I_k as $I_k \triangleq \{a_{n,k} \mid a_{n,k} \leq \psi^k\}$, and proceed to bound $|I_k|$.

Lemma 5. *For any $k \geq 1$ we have $|I_k| \geq \epsilon^{-1} \ln(\psi\epsilon/4) - 1$.*

Proof. Clearly, the cardinality of I_k is the largest n for which $a_{n,k} \leq \psi^k$. According to Lemma 5, we have that $a_{n,k} \leq 4\epsilon^{-1}(1 + \epsilon)^{n+1}\psi^{k-1}$, and thus:

$$\begin{aligned} |I_k| &= \arg \max \{n \mid 4\epsilon^{-1}(1 + \epsilon)^{n+1}\psi^{k-1} \leq \psi^n\} \\ &\geq \log_{1+\epsilon}(\psi\epsilon/4) - 1 = \frac{\ln(\psi\epsilon/4)}{\ln(1 + \epsilon)} - 1 \geq \epsilon^{-1} \ln(\psi\epsilon/4) - 1. \end{aligned} \quad \square$$

Finally, we show that the slack is not zero (i.e., $\frac{2}{W} \leq \tau$), we get a new lower bound:

Lemma 6. *For $\max\left\{\frac{2}{W}, \sqrt{\frac{4}{RW}}\right\} \leq \tau \leq 1$, any deterministic algorithm \mathbb{A} requires at least $\Omega\left(\tau^{-1}(\log \epsilon^{-1} + \log \log \log(RW\tau))\right)$ bits to solve (W, τ, ϵ) -MULTIPLICATIVE SUMMING.*

Proof. We use $\text{rep}(x) \triangleq (x \bmod R) \cdot R^{\lfloor x/R \rfloor}$ to denote a sequence in $\{\sigma R^* \mid \sigma \in [R]\}$ that has a sum of x . For an integer set I_k , we denote $\text{rep}(I_k) \triangleq \{\text{rep}(x) \mid x \in I_k\}$. We now choose the value of ψ to be $\psi \triangleq \log_{\lceil \tau^{-1}/2 \rceil}(R \lfloor W/8 \rfloor)$; notice that $\psi \geq 2$ as required. Next, consider the language:

$$\begin{aligned} \overline{L_{M,2}} &\triangleq 0^W \cdot 0^{W\tau} \cdot \text{rep}(I_{\lceil \tau^{-1}/2 \rceil}) \cdot 0^{W\tau} \cdot \text{rep}(I_{\lceil \tau^{-1}/2 \rceil - 1}) \cdots 0^{W\tau} \cdot \text{rep}(I_1) \\ &= \left\{ 0^w \cdot w_1 \cdot w_2 \cdots w_{\lceil \tau^{-1}/2 \rceil} \mid \forall i : w_i \in \left\{ 0^{W\tau} \cdot \text{rep}(x) \mid x \in I_{\lceil \tau^{-1}/2 \rceil + 1 - i} \right\} \right\}. \end{aligned}$$

That is, every word in $\overline{L_{M,2}}$ consists of concatenation of words $w_1, \dots, w_{\lceil \tau^{-1}/2 \rceil}$, such that every w_i starts with $W\tau$ zeros, and then with a string representation an integer in $I_{\lceil \tau^{-1}/2 \rceil + 1 - i}$ which is defined above. According to Lemma 5 we have that

$$\begin{aligned} \log(|\overline{L_{M,2}}|) &\geq \log\left((\epsilon^{-1} \ln(\psi\epsilon/4) - 1)^{\lceil \tau^{-1}/2 \rceil}\right) = \lceil \tau^{-1}/2 \rceil (\log \epsilon^{-1} + \log \log(\psi\epsilon) - O(1)) \\ &= \Omega\left(\tau^{-1} \left(\log \epsilon^{-1} + \log \log(\log_{\lceil \tau^{-1}/2 \rceil}(R \lfloor W/8 \rfloor \epsilon))\right)\right) \\ &= \Omega\left(\tau^{-1} (\log \epsilon^{-1} + \log \log(\log(RW\epsilon) - \log(\tau^{-1})))\right) = \Omega\left(\tau^{-1} (\log \epsilon^{-1} + \log \log \log(RW\tau))\right) \end{aligned}$$

Next, we show that every two words in $\overline{L_{M,2}}$ must reach different memory configurations, thereby implying a $\Omega(\log(|\overline{L_{M,2}}|))$ bits lower bound. Let $S_1 \neq S_2 \in \overline{L_{M,2}}$ such that $S_1 = 0^W \cdot w_{1,1} \cdots w_{\lceil \tau^{-1}/2 \rceil,1}$, $S_2 = 0^W \cdot w_{1,2} \cdots w_{\lceil \tau^{-1}/2 \rceil,2}$, and $\forall i \in \{1, \dots, \lceil \tau^{-1}/2 \rceil\} j \in \{1, 2\} : w_{i,j} \in \{0^{W\tau} \cdot \text{rep}(x) \mid x \in I_{\lceil \tau^{-1}/2 \rceil + 1 - i}\}$. We next assume by contradiction that S_1 and S_2 leads \mathbb{A} to the same memory configuration. Let $\chi \in \{1, \dots, \lceil \tau^{-1}/2 \rceil\}$ such that $w_{\chi,1} \neq w_{\chi,2}$. Since \mathbb{A} reaches an identical configuration after reading S_1, S_2 , and as it is deterministic, \mathbb{A} must reach the same configuration when processing $S_1 \cdot 0^{2W\tau(\chi-1)}$ and $S_2 \cdot 0^{2W\tau(\chi-1)}$. Next, observe that for every $k \in \{1, \dots, \lceil \tau^{-1}/2 \rceil\}$, the representation length of any of its words is bounded by $\lceil \psi^k/R \rceil$. Thus, the length of a word in

$$\{w_1 \cdot w_2 \cdots w_{\lceil \tau^{-1}/2 \rceil} \mid \forall i : w_i \in \{0^{W\tau} \cdot \text{rep}(x) \mid x \in I_{\lceil \tau^{-1}/2 \rceil + 1 - i}\}\}$$

is at most

$$\begin{aligned} \sum_{k=1}^{\lceil \tau^{-1}/2 \rceil} \lceil W\tau + \psi^k/R \rceil &\leq \lceil \tau^{-1}/2 \rceil (W\tau + 1) + 2\psi^{\lceil \tau^{-1}/2 \rceil}/R \\ &= \lceil \tau^{-1}/2 \rceil (W\tau + 1) + 2 \lfloor W/8 \rfloor \leq 3W/4 + \lceil \tau^{-1}/2 \rceil + W\tau \leq W + W\tau. \end{aligned}$$

Now, since every word $w_{i,j}$ starts with a sequence of $W\tau$ zeros, the slack size chosen by the algorithm is irrelevant and the sums the algorithm must estimate are $\sum_{i=\chi}^{\lceil \tau^{-1}/2 \rceil} s(w_{i,1})$ and $\sum_{i=\chi}^{\lceil \tau^{-1}/2 \rceil} s(w_{i,2})$, where $s(w_{i,j})$ is simply the sum of the symbols in $w_{i,j}$. Note that $s(w_{\chi,1})$ and $s(w_{\chi,2})$ are integers in $I_{\lceil \tau^{-1}/2 \rceil + 1 - \chi}$. We assume without loss of generality that $s(w_{\chi,1}) < s(w_{\chi,2})$ (i.e., $s(w_{\chi,1}) < s(w_{\chi,2}) \in I_{\lceil \tau^{-1}/2 \rceil + 1 - \chi}$). Finally, it follows that

$$\sum_{i=\chi}^{\lceil \tau^{-1}/2 \rceil} s(w_{i,1}) \leq s(w_{\chi,1}) + \sum_{i=\chi+1}^{\lceil \tau^{-1}/2 \rceil} \max(I_{\lceil \tau^{-1}/2 \rceil + 1 - i}) \leq s(w_{\chi,1}) + \sum_{k=1}^{\chi-1} \psi^k \leq \frac{s(w_{\chi,2})}{1 + \epsilon},$$

where the last inequality follows from the definition of $I_{\lceil \tau^{-1}/2 \rceil + 1 - \chi}$. Thus, no \hat{S} value is correct for both $S_1 \cdot 0^{2W\tau(\chi-1)}$ and $S_2 \cdot 0^{2W\tau(\chi-1)}$. \square

Finally, we combine Lemma 3 and Lemma 6 to obtain the following lower bound:

Theorem 3. *For $\epsilon < 1/4$, $\max\left\{\frac{2}{W}, \sqrt{\frac{4}{RW}}\right\} \leq \tau \leq 1$, any deterministic algorithm \mathbb{A} for the (W, τ, ϵ) -MULTIPLICATIVE SUMMING problem requires at least $\Omega(\log(W/\epsilon) + \log \log(RW\epsilon) + \tau^{-1}(\log \epsilon^{-1} + \log \log \log(RW\tau)))$ bits.*

4 Upper Bounds

In this section we introduce algorithms for the SLACKY SUMMING problems.

4.1 (W, τ) -EXACT SUMMING

We divide the stream into equal $W\tau$ -sized blocks and sum the number of arriving elements in each block with a $\lceil \log(RW\tau + 1) \rceil$ bits counter. The sum of the current block is maintained in a variable called y , c maintains the number of elements within the current block and i is the current block number. The variable b is a cyclic buffer of τ^{-1} blocks. Every $W\tau$ steps, we assign the value of y to the oldest block (b_i) and increment i . Intuitively, we “forget” b_i as its block is no longer part of the window. To satisfy queries in constant time, we also maintain the sum of all active counters in a $\lceil \log(RW(1 + \tau) + 1) \rceil$ -bits variable named B . Algorithm 1 provides pseudocode for the described algorithm.

Algorithm 1 (W, τ) -EXACT SUMMING Algorithm

```

Initialization:  $y = 0, b = \bar{0}, B = 0, i = 0, c = 0$ .
1: function UPDATE( $x$ )
2:    $y \leftarrow y + x$ 
3:    $c \leftarrow (c + 1) \bmod W\tau$ 
4:   if  $c = 0$  then ▷ End of block
5:      $B \leftarrow B - b_i + y$ 
6:      $b_i \leftarrow y$ 
7:      $y \leftarrow 0$ 
8:      $i \leftarrow (i + 1) \bmod \tau^{-1}$ 
9: function OUTPUT
10:  return  $\langle B + y, c \rangle$ 

```

We now analyze the memory consumption of Algorithm 1.

Theorem 4. *Algorithm 1 uses $(\tau^{-1} + 1) \lceil \log(RW\tau + 1) \rceil + \log(RW^2) + O(1)$ bits.*

Proof. y takes $\lceil \log(RW\tau + 1) \rceil$ bits; B requires $\lceil \log(RW + 1) \rceil$; i adds $\lceil \log \tau^{-1} \rceil$ bits, while c needs $\lceil \log W\tau \rceil$ bits. Finally, b is a τ^{-1} -sized array of counters, each allocated with $\lceil \log(RW\tau + 1) \rceil$ bits. Overall, Algorithm 1 uses at most $(\tau^{-1} + 1) \lceil \log(RW\tau + 1) \rceil + \log(RW^2) + 4$ bits. \square

We conclude that Algorithm 1 is asymptotically optimal.

Theorem 5. *Let $\mathcal{B} \triangleq \max \{ \lfloor \log(RW^2) \rfloor, \lceil \tau^{-1}/2 \rceil \log(RW\tau + 1) \}$ be the (W, τ) -EXACT SUMMING lower bound of Theorem 1. Algorithm 1 uses at most $\mathcal{B}(4 + o(1))$ memory bits.*

Proof. As shown in Theorem 4, the number of bits used by Algorithm 1 is

$$\begin{aligned}
 & (\tau^{-1} + 1) \lceil \log(RW\tau + 1) \rceil + \log(RW^2) + O(1) \\
 & \leq \tau^{-1} \lceil \log(RW\tau + 1) \rceil + 2 \log(RW^2) + O(1) \leq \mathcal{B}(4 + o(1)) \quad \square
 \end{aligned}$$

Theorem 5 shows that Algorithm 1 is only x4 larger than the lower bound. We now show that in some special cases we can get considerably closer to the lower bound.

Theorem 6. *Consider a stream where $R = O(1)$ and $\tau = 1$. There exists a (W, τ) -EXACT SUMMING algorithm that uses $1.5\mathcal{B} + O(1)$ bits, where \mathcal{B} is the lower bound.*

Proof. Our method here is similar to Algorithm 1, but the constant τ value allows us to compute $\sum_{i=1}^{\tau^{-1}} b_i = b_0 + b_1$ in $O(1)$ without tracking it in B . Thus, our algorithm only requires $3 \log W + O(1)$ bits, while Theorem 1 gives a lower bound of $\lfloor 2 \log W \rfloor$. \square

Finally, in Appendix C we show that Algorithm 1 is correct.

4.2 (W, τ, ϵ) -ADDITIVE SUMMING

We now show that additional memory savings can be obtained by combining slackness with additive error. First, we consider the case where $\tau \leq 2\epsilon$. In [4], we proposed an algorithm that sums over (exact) W elements window using the optimal $\Theta(\epsilon^{-1} + \log W)$ bits, with an additive error of $RW\epsilon$. Next, notice that if an algorithm solves (W, τ, ϵ) -ADDITIVE SUMMING, it also solves $(W, \tau, \tau/2)$ -ADDITIVE SUMMING; hence, we can apply Theorem 2 to conclude that it requires $\Omega(\tau^{-1} + \log W) = \Omega(\epsilon^{-1} + \log W)$. Thus, we can run the algorithm from [4] and remain asymptotically memory optimal with no slack at all!

Henceforth, we assume that $\tau > 2\epsilon$; we present an algorithm for the problem using a technique of *2-pass rounding*. Whenever a new item arrives, we scale it by R and then round the results to $O(\log \epsilon^{-1})$ bits. As in Section 4.1, we break the stream into non-overlapping blocks of size $W\tau$ and compute the sum of each block separately. However, we now sum the rounded values rather than the exact input, with a $O(\log \frac{W\tau}{\epsilon})$ -bits counter denoted y . Once the block is completed, we *round its sum* such that it is represented with $O(\log \frac{\tau}{\epsilon})$ bits. Note that this second rounding is done for the entire block's sum while we still have the “exact” sum of rounded fractions. Therefore, we propagate the second rounding error to the following block. An illustration of our algorithm appears in Figure 3. Here, $\text{Round}_v(z)$ refers to rounding a fractional number $z \in [0, 1]$ into the closest number \tilde{z} such that $2^v \cdot \tilde{z}$ is an integer. Algorithm 2 provides pseudo code for the described algorithm. The code uses the following variables:

1. y - a fixed point variable that uses $\lceil \log W\tau \rceil + 1$ bits to store its integral part and another $v_1 \triangleq \lceil \log \epsilon^{-1} \rceil + 1$ bits for storing the fractional part.
2. b - a cyclic array that contains τ^{-1} elements, each of which takes $v_2 \triangleq \lceil \log \frac{\tau}{\epsilon} \rceil$ bits.
3. B - keeps the sum of elements in b and is represented using $\log(\tau^{-1} \lceil \log \frac{\tau}{\epsilon} \rceil + 1)$ bits.
4. i - the index variable used for tracking the oldest block in b .
5. c - a variable that keeps the offset within the $W\tau$ sized block.

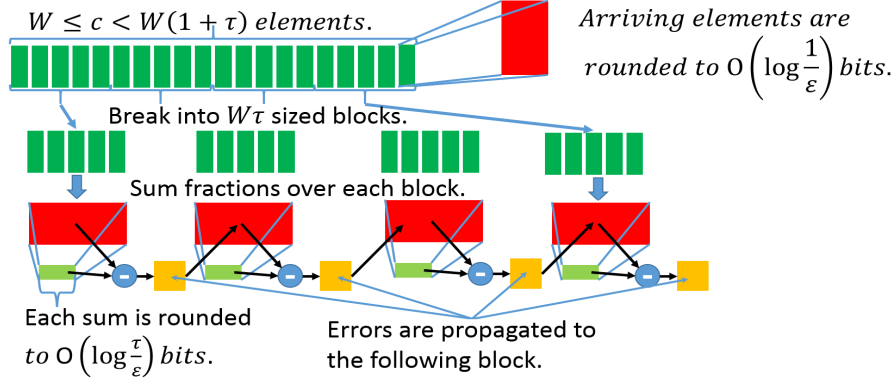


Figure 3: An illustration of our 2-pass rounding technique. Each element is rounded upon arrival to $(\lceil \log \epsilon^{-1} \rceil + 1)$ bits. We then sum the rounded fractions of each block and round the resulting sum into $\lceil \log \frac{\tau}{\epsilon} \rceil$ bits. The second rounding error is propagated to the successive block.

We now analyze the memory consumption of Algorithm 2.

Theorem 7. *Algorithm 2 uses $\tau^{-1} \log(\frac{\tau}{\epsilon})(1 + o(1)) + 2 \log(W/\epsilon)$ bits.*

Algorithm 2 (W, τ, ϵ) -ADDITIVE SUMMING Algorithm

Initialization: $y = 0, b = 0, B = 0, i = 0, c = 0$.

```

1: function UPDATE( $x$ )
2:    $x' \leftarrow \text{Round}_{v_1}(\frac{x}{R})$   $\triangleright \text{Round}(\frac{x}{R})$  such that  $x' \cdot 2^{v_1} \in \mathbb{N}$ 
3:    $y \leftarrow y + x'$ 
4:    $c \leftarrow (c + 1) \bmod W\tau$ 
5:   if  $c = 0$  then  $\triangleright$  End of block
6:      $B \leftarrow B - b_i$ 
7:      $b_i \leftarrow \text{Round}_{v_2}(\frac{y}{W\tau})$   $\triangleright$  Replace the value for the block that has left the window.
8:      $B \leftarrow B + b_i$ 
9:      $y \leftarrow y - W\tau \cdot b_i$ 
10:     $i \leftarrow (i + 1) \bmod \tau^{-1}$ 
11: function OUTPUT
12:   return  $\langle R \cdot (W\tau \cdot B + y), c \rangle$ 

```

Proof. y requires $\log(\frac{W\tau}{\epsilon}) + O(1)$ bits; b requires another $\tau^{-1} \lceil \log(\frac{\tau}{\epsilon}) \rceil$; B takes additional $\log(\tau^{-1} \lceil \log \frac{\tau}{\epsilon} \rceil + 1)$ bits; i adds $\lceil \log \tau^{-1} \rceil$ bits, while c is represented with $\lceil \log W\tau \rceil$ bits. Overall, the space requirement is $\tau^{-1} \lceil \log(\frac{\tau}{\epsilon}) \rceil (1 + o(1)) + 2 \log(W/\epsilon)$ bits. \square

Corollary 1. Let $\mathcal{B} \triangleq \max \{ \log(W/\epsilon) - O(1), \lceil \tau^{-1}/2 \rceil \log \lfloor \tau/2\epsilon + 1 \rfloor \}$ be the (W, τ, ϵ) -ADDITIVE SUMMING space lower bound of Theorem 2, then Algorithm 2 uses $\mathcal{B} \cdot (4 + o(1))$ bits.

Next, we prove that our algorithm provides the requested error guarantee.

Theorem 8. Algorithm 2 solves the (W, τ, ϵ) -ADDITIVE SUMMING problem.

Proof. First, observe that at all times $c \in [W\tau - 1]$ as needed. Denote the stream by $S = x_1, x_2, \dots, x_{t+c}$, such that c represents the number of elements within the current block. Our goal is to show that Algorithm 2 provides a $RW\epsilon$ approximation to the sum of the last $W + c$ elements $(x_{t-W+1} \dots x_{t+c})$. That is, the quantity we approximate is

$$S \triangleq \sum_{\ell=1}^{W+c} x_{t-W+\ell}.$$

For any $\ell \in [t + c]$, we use y_ℓ to denote the value of the variable y after the ℓ 'th item was added. Note that within a block, y simply sums the rounded scaled inputs; whenever a block ends, we reduce the value of y by $W\tau \cdot \text{Round}_{v_2}(\frac{y}{W\tau})$, but make up for it by setting b_i . Further, when processing $x_{t-W+1} \dots x_{t+c}$, we replace all of the values of b that were determined before the last $W + c$ elements, and none of the set value leaves the window by time $t + c$. That is, i reaches every value in $[\tau^{-1}]$ exactly once throughout the last $W + c$ updates. This gives us the following equality:

$$y_{t-W} + \sum_{i=1}^{W+c} x'_{t-W+i} = W\tau \sum_{i=1}^{\tau^{-1}} b_i + y_{t+c} = W\tau \cdot B + y_{t+c}.$$

Thus, we can express the algorithm's estimate of the sum value $\hat{S} \triangleq R \cdot (W\tau \cdot B + y_{t+c})$ as:

$$\hat{S} = R \cdot \left(y_{t-W} + \sum_{i=1}^{W+c} x'_{t-W+i} \right). \quad (1)$$

Next, notice that since $\forall \ell : x'_\ell = \text{Round}_{v_1}(\frac{x_\ell}{R})$, we have $|x'_\ell - \frac{x_\ell}{R}| \leq 2^{-v_1-1}$ and thus:

$$\left| \sum_{\ell=1}^{W+c} x'_{t-W+\ell} - \frac{1}{R} \sum_{\ell=1}^{W+c} x_{t-W+\ell} \right| = \left| \sum_{\ell=1}^{W+c} x'_{t-W+\ell} - \frac{1}{R} S \right| \leq (W+c) \cdot 2^{-v_1-1}. \quad (2)$$

Also, since we assumed that x_{t-W} was the last of a $W\tau$ -sized block, we know that the value of y is bounded, and specifically:

$$|y_{t-W}| \leq W\tau \cdot 2^{-v_2-1}. \quad (3)$$

Plugging (2) and (3) into (1), we get a bound on the error:

$$\mathcal{E} \triangleq |S - \hat{S}| \leq R \cdot (W\tau \cdot 2^{-v_2-1} + (W + c) \cdot 2^{-v_1-1}) < RW \cdot (\tau 2^{-v_2-1} + 2^{-v_1}).$$

Thus, since $v_1 = \lceil \log \epsilon^{-1} \rceil + 1$, $v_2 = \lceil \log \frac{\tau}{\epsilon} \rceil$, we get the desired $\mathcal{E} < RW\epsilon$ error bound and conclude that Algorithm 2 solves (W, τ, ϵ) -ADDITIVE SUMMING. \square

4.3 (W, τ, ϵ) -MULTIPLICATIVE SUMMING

Next, we present an algorithm for a $(1+\epsilon)$ multiplicative approximation of the SLACKY SUMMING problem. Compared to Algorithm 1, we achieve a space reduction by representing each sum of $W\tau$ elements using $O(\log \log (RW\tau) + \log \epsilon^{-1})$ bits. Specifically, when a block ends, if its sum was y , we store $\rho = \lfloor \log_{(1+\epsilon/2)} y \rfloor$ (we allow a value of $-\infty$ for ρ if $y = 0$). To achieve $O(1)$ OUTPUT, we also store an approximate window sum B , which is a *fixed point* fractional variable with $O(\log RW)$ bits for its integral part and additional $O(\log \epsilon^{-1})$ bits for storing a fraction. In order to update B 's value for a new ρ , we *round down* the value of $(1+\epsilon)^\rho$. Specifically, for a real number x , we denote $(x)_\downarrow \triangleq \lfloor x \cdot k \rfloor / k$, for $k \triangleq \lceil \frac{4}{\epsilon} \rceil$. Our pseudo code appears in Algorithm 3. Algorithm 3 requires $O(\tau^{-1} (\log \log (RW\tau) + \log \epsilon^{-1}) + \log RW)$ bits of space and is memory optimal when $R = W^{O(1)}$ and $\tau = \Omega\left(\frac{\log \log W}{\log W}\right)$, which includes constant τ values. The full analysis of Algorithm 3 is deferred to Appendix D.

Algorithm 3 (W, τ, ϵ) -MULTIPLICATIVE SUMMING Algorithm

Initialization: $y = 0, b = \bar{0}, B = 0, i = 0, c = 0$.

```

1: function UPDATE( $x$ )
2:    $y \leftarrow y + x$ 
3:    $c \leftarrow (c + 1) \bmod W\tau$ 
4:   if  $c = 0$  then
5:      $\rho \leftarrow \lfloor \log_{(1+\epsilon/2)} y \rfloor$ 
6:      $B \leftarrow B - ((1+\epsilon/2)^{b_i})_\downarrow + ((1+\epsilon/2)^\rho)_\downarrow$ 
7:      $b_i \leftarrow \rho$ 
8:      $y \leftarrow 0$ 
9:      $i \leftarrow (i + 1) \bmod \tau^{-1}$ 
10: function OUTPUT
11:   return  $\langle B + y, c \rangle$ 

```

▷ End of block
▷ If $y = 0$ we use $\rho = -\infty$ and $(1+\epsilon/2)^\rho = 0$

4.4 The Mean of a Slacky Window

For some applications there is value in knowing the *mean* of a slacky window. For example, a load balancer may be interested in the average transmission throughput. In exact windows, the sum and the mean can be derived from each other as the window size is constant. However, in slacky windows the window size changes. Our algorithms also return the current slack offset $0 \leq c < W\tau$ which enables the computation of the mean. That is, by dividing \hat{S} with $W + c$ we get an estimation of the mean (we assume that stream size is larger than W). Specifically, Algorithm 1 provides the exact mean; Algorithm 2 approximates it with $R\epsilon$ additive error, while Algorithm 3 yields a $(1+\epsilon)$ multiplicative approximation.

5 Counting Distinct Items on Slacky Windows

Estimating the number of **distinct** elements in a stream is another useful metric. In networking, the packet header is used to identify different flows and it is useful to know how many distinct flows are currently active. A sudden spike in the number of active flows is often an indication of a threat to the network. It may indicate the propagation of a worm or virus, port scans that are used to detect vulnerabilities in the system and even *Distributed Denial of Service (DDoS)* attacks [12, 20].

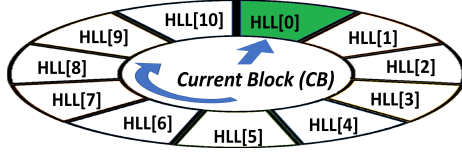
Accurately counting distinct elements requires linear space [22]. Intuitively, one needs to maintain a list of all previously encountered identifiers. Therefore, accurate measurement does not scale to large streams and approximate solutions are very popular. Specifically, count distinct algorithms often use randomized estimators [3, 13, 16, 21]. Randomized algorithms typically use a hash function $H : \mathbb{D} \rightarrow \{0, 1\}^\infty$ that maps ids to infinite bit strings. When a maximal cardinality bound is known, finite strings are used and typically 32 bit integers suffice to reach estimations of over 10^9 [22]. We assume that the hashed values are distributed uniformly at random, i.e., $\forall d \in \mathbb{D} : \Pr[H(d)_i] = 0.5$.

Count distinct algorithms look for certain *observables* in the hashes. For example, some algorithms [3, 24] look at the minimal observed hash value as a real number in $[0, 1]$ and exploit that $\mathbb{E}(\min(H(\mathcal{M}))) = \frac{1}{n+1}$, where n is the number of the distinct items in the multi-set \mathcal{M} . Another possibility is to look for patterns of the form $0^{\beta-1}1$ [16, 22]. When such a pattern is first encountered, it is likely that there were at least 2^β unique elements.

Monitoring observables significantly reduces the required amount of space. The challenge is the variability as a single counter instance is often not accurate enough. The variability can be reduced by performing m independent experiments. However, such a simplistic strategy has computational overheads as it requires calculating m different hash functions. More importantly, such a technique requires a large family of independent hash functions for which no construction is currently known [2]. Instead, *stochastic averaging* [21] is used to mimic the effects of multiple experiments with a single hash function. To do that, we use some of the bits in the hashed value to map each identifier to one of the instances. That way, each separate instance counts a distinct set of ids and summing the distinct value of all instances yields the number of distinct elements in the stream. It is shown in [21] that averaging these separate instances yields similar results to m independent experiments. Specifically, the standard deviation is reduced by a factor of $\frac{1}{\sqrt{m}}$ and enables algorithms to reach high accuracy with minor computational overheads. The state of the art count distinct algorithm is *HyperLogLog (HLL)* [22], which is being used in multiple Google projects [25]. HLL requires m bytes and its standard deviation is $\sigma \approx \frac{1.04}{\sqrt{m}}$. HLL was extended to sliding windows by [10, 31] and the method is used to detect port scans in networked systems [11]. *Sliding Window HLL (W-HLL)* requires $5m \ln(W/m)$ bytes and its standard deviation is $\sigma \approx \frac{1.04}{\sqrt{m}}$. In this work, we present τ -*Slacky HyperLogLog* (τ -*SHLL*) that requires $(\tau^{-1} + 1)m$ bytes and its standard deviation is $\sigma \approx \frac{1.04}{\sqrt{m}}$. When τ is constant, Slacky HLL requires $O(m)$ words. When $\tau^{-1} = o(\ln(W/m))$ it requires asymptotically less space than W-HLL. For completeness, we provide an overview of the HyperLogLog algorithm in Appendix E.

5.1 τ -Slacky HyperLogLog

We now present the τ -*SHLL* algorithm. We logically divide the stream into fixed $W\tau$ sized blocks. Our algorithm maintains a cyclic buffer of $\tau^{-1} + 1$ HLL instances. Each instance has a buffer index in the range $(0, \tau^{-1})$ and the symbol $HLL[i]$ refers to the HLL instance at index i , and $HLL[i]_k$ denotes its k 'th register for $k \in [m-1]$. We use two counters: *Current Block (CB)* holds values in $\{0, 1, \dots, \tau^{-1}\}$ and *Place in Block (PB)* counts how many items are included in the current block. Initially, CB and PB are set to 0 and new items are always added to $HLL[CB]$.



Parameters:

Window Size (W) = 1,000,000
Slack (τ) = 0.1
Block Size (τW) = 100,000

Variables:

Current Block (CB): 0
Place in Block (PB): 0

Figure 4: An example of the τ -Slacky HLL algorithm with a window of size W and $\tau = 0.1$. τ -SHLL relies on $\tau^{-1} + 1 = 11$ HLL instances, each is used to count 100,000 events. Every $W\tau$ events we increment the CB counter and reset $HLL[CB]$. This enables our algorithm to work here on window sizes (including slack) of minimum 1,000,000 events and maximum 1,100,000 events.

Each additional item increments PB and once PB reaches the value $W\tau$, we set $PB = 0$ and increment CB . We also reset the HLL instance at $HLL[CB]$ by setting all its registers to $-\infty$. Doing so enables us to forget information that is guaranteed not to be in the window. To query Slacky HLL, we use the maximum of each of the m registers to generate Z and continue as in HLL. A pseudo code of Slacky HLL is found in Algorithm 4 and an example of the algorithm's setup is illustrated in Figure 4. Note that α_m is range correction constant that depends on m .

Algorithm 4 τ -Slacky HyperLogLog (for slacky windows)

Initialization: $CB \leftarrow 0, PB \leftarrow 0$. For $i \in \{0, 1, \dots, \tau^{-1}\}$, Initialize $HLL[i]$.

- 1: **function** UPDATE(x)
- 2: $HLL[CB].Update(x)$
- 3: $PB \leftarrow (PB + 1) \bmod W\tau$
- 4: **if** $PB \leftarrow 0$ **then** \triangleright End of block
- 5: $CB \leftarrow (CB + 1) \bmod (\tau^{-1} + 1)$
- 6: Initialize $HLL[CB]$
- 7: **function** QUERY(x)
- 8: For every $k \in \{0, 1, \dots, m-1\}$, compute $M_{max}[k] \triangleq \max \{HLL[i]_k \mid i \in [\tau^{-1} + 1]\}$.
- 9: $Z \leftarrow \left(\sum_{j=0}^{m-1} 2^{-M_{max}[j]} \right)^{-1}$
- 10: **return** $\langle \alpha_m \cdot m^2 \cdot Z, PB \rangle$

The next theorem, whose proof is deferred to Appendix F, shows that τ -SHLL is correct.

Theorem 9. Let $\langle \hat{D}, PB \rangle$ be the query result of Slacky HLL, then \hat{D} is the query result of HLL for a stream containing the last $W + PB$ events (for streams longer than W).

6 Discussion

In this work, we investigated the space requirements of slacky sliding window problems. The size of such windows can be dynamically adjusted between W and $W(1 + \tau)$, where τ is a small positive parameter. Specifically, we studied the SLACKY SUMMING problem, where a slacky window sum is maintained. We introduced space lower bounds on deterministically computing the exact sum, as well as additive and multiplicative approximations.

Specifically, we showed an $\Omega(\tau^{-1} \log(RW\tau))$ bits lower bound for the exact summing problem, an $\Omega(\tau^{-1} \log \lfloor 1 + \tau/\epsilon \rfloor + \log(W/\epsilon))$ bound for $RW\epsilon$ additive approximations, and an $\Omega(\log(W/\epsilon) + \log \log R + \tau^{-1} (\log \epsilon^{-1} + \log \log \log(RW\tau)))$ bound for $(1 + \epsilon)$ multiplicative approximations.

For all settings, we provide deterministic memory optimal algorithms that operate in worst case constant time. Our exact and additive summing algorithms are asymptotically optimal for any τ, ϵ, W . The multiplicative algorithm requires $O(\tau^{-1} (\log \log(RW\tau) + \log(\epsilon^{-1})) + \log(RW))$ bits and is asymptotically optimal for constant τ and $R = \text{poly}(W)$.

Finally, we showed that slacky windows are useful also for the count distinct problem that estimates the number of distinct ids in a stream. We extended the popular HyperLogLog (HLL)

algorithm to slacky windows and demonstrated asymptotic improvement compared to accurate sliding windows when τ is sub logarithmic in W . In the future, we would like to explore the impact of randomization on the space required for these problems.

References

- [1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. *ACM SIGCOMM* 2014.
- [2] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of ACM STOC*, 1996.
- [3] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *RANDOM*, 2002.
- [4] Ran Ben Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Efficient Summing over Sliding Windows. In *SWAT*, 2016.
- [5] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM*, 2016.
- [6] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Optimal elephant flow detection. In *IEEE INFOCOM*, 2017.
- [7] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *IEEE INFOCOM*, 2017.
- [8] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *ACM CoNEXT*, 2011.
- [9] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters, 2006.
- [10] Y. Chabchoub and G. Hebrail. Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window. In *2010 IEEE ICDM Workshops*, 2010.
- [11] Yousra Chabchoub, Raja Chiky, and Betul Dogan. How can sliding hyperloglog and ewma detect port scan attacks in ip traffic? *EURASIP Journal on Information Security*, 2014.
- [12] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. In *ACM CSUR*, 2007.
- [13] Philippe Chassaing and Lucas Gerin. Efficient estimation of the cardinality of large data sets. In *Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities*, 2006.
- [14] Min Chen and Shigang Chen. Counter tree: A scalable counter architecture for per-flow traffic measurement. In *IEEE ICNP*, 2015.
- [15] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal of Computing*.
- [16] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *ESA*, 2003.
- [17] G. Einziger and R. Friedman. TinyLFU: A highly efficient cache admission policy. In *PDP 2014*.

- [18] Gil Einziger, Benny Fellman, and Yaron Kassner. Independent counter estimation buckets. In *IEEE INFOCOM*, 2015.
- [19] Gil Einziger and Roy Friedman. Counting with TinyTable: Every Bit Counts! In *ICDCN 2016*.
- [20] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *ACM IMC*, 2003.
- [21] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 1985.
- [22] Philippe Flajolet, ric Fusy, Olivier Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.
- [23] Pedro Garcia-Teodoro, Jess E. Daz-Verdejo, Gabriel Maci-Fernndez, and E. Vzquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers and Security*, 2009.
- [24] Frdric Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 2009.
- [25] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *ACM EDBT*, 2013.
- [26] Nan Hua, Bill Lin, Jun (Jim) Xu, and Haiquan (Chuck) Zhao. Brick: A novel exact active statistics counter architecture. In *ACM/IEEE ANCS*, 2008.
- [27] Abdul Kabbani, Mohammad Alizadeh, Masato Yasuda, Rong Pan, and Balaji Prabhakar. Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. In *IEEE HOTI*, 2010.
- [28] Yang Liu, Wenji Chen, and Yong Guan. Near-optimal approximate membership query over time-decaying windows. In *Proceedings of IEEE INFOCOM*, pages 1447–1455, April 2013.
- [29] B. Mukherjee, L.T. Heberlein, and K.N. Levitt. Network intrusion detection. *Network, IEEE*, 1994.
- [30] Moni Naor and Eylon Yogev. Sliding bloom filters. In *Algorithms and Computation*. 2013.
- [31] ric Fusy and Frcrie Giroire. *Estimating the number of Active Flows in a Data Stream over a Sliding Window*, pages 223–231. 2007.
- [32] Erez Tsidon, Iddo Hanniel, and Isaac Keslassy. Estimators also need shared values to grow together. In *IEEE INFOCOM*, 2012.
- [33] Hao Wang, H. Zhao, Bill Lin, and Jun Xu. Dram-based statistics counter array architecture with performance guarantee. *IEEE/ACM Transactions on Networking*, 20(4), Aug 2012.
- [34] Li Yang, Wu Hao, Pan Tian, Dai Huichen, Lu Jianyuan, and Liu Bin. Case: Cache-assisted stretchable estimator for high speed per-flow measurement. In *IEEE INFOCOM*, 2016.
- [35] L. Ying, R. Srikant, and X. Kang. The power of slightly more than one sample in randomized load balancing. In *IEEE INFOCOM*, 2015.

A Proof of Lemma 1

Proof. Consider the following language

$$L_{E_1} \triangleq \{0^{W\tau+i}\sigma R^{W-i-1}0^j \mid i, j \in [W-1], i \geq j, \sigma \in ([R] \setminus \{0\})\} \cup \{0^{W+W\tau}\}$$

That is, L_{E_1} contains a word with $W + W\tau$ consecutive zeros and the rest of the words in L_{E_1} are composed of these components in this order:

- $W\tau + i$ zeros for some $i \in [W-1]$.
- a non zero symbol σ .
- $W - i - 1$ repetitions of the maximal symbol (R).
- j zeros for some $j \in [i]$.

Our lower bounds stems from the observation that every word in L_{E_1} must lead to a different state. The language size is: $|L_{E_1}| = 1 + \sum_{i=0}^{W-1} R(i+1) = 1 + RW(W+1)/2$. Therefore, the number of required bits is at least: $\lceil \log |L_{E_1}| \rceil > (\log(RW^2) - 1)$. Further, since this number is an integer then at least $\lfloor \log(RW^2) \rfloor$ bits are required.

First, notice that the word composed of $W + W\tau$ zeros requires a unique configuration as \mathbb{A} must return 0 after processing that word. In contrast, it must not return 0 after processing any other word as there is at least a single R within last W elements.

Let $w_1, w_2 \in L_{E_1}$ be two different words. We need to show that w_1 and w_2 require different memory configuration.

By definition of L_{E_1} , $w_1 = 0^{W\tau+i_1}\sigma_1 R^{W-i_1-1}0^{j_1}$ and $w_2 = 0^{W\tau+i_2}\sigma_2 R^{W-i_2-1}0^{j_2}$. Observe that the last W elements of w_1, w_2 are $0^{i_1-j_1}\sigma_1 R^{W-i_1-1}0^{j_1}$ and $0^{i_2-j_2}\sigma_2 R^{W-i_2-1}0^{j_2}$ respectively and that both are preceded with a $W\tau$ zeros. If $i_1 \neq i_2$ or $\sigma_1 \neq \sigma_2$, then $\sigma_1 + R \cdot (W - i_1 - 1) \neq \sigma_2 + R \cdot (W - i_2 - 1)$ and thus \mathbb{A} cannot return the same count for both, regardless of slack. Note that the slack is all zeros in both W_1 and W_2 .

Next, assume that $i_1 = i_2$, $\sigma_1 = \sigma_2$ and that without loss of generality $j_1 < j_2$. This means that both w_1 and w_2 have the same count.

Since $j_1 < j_2$, w_1 is a strict prefix of w_2 , i.e., $w_2 = w_1 \cdot 0^{j_2-j_1}$. Assume by contradiction that after processing w_1, w_2 \mathbb{A} reaches the same memory configuration. Since \mathbb{A} is deterministic, this means that it must reach the same configuration after seeing $w_1 \cdot 0^{z(j_2-j_1)}$ for any integer z . By choosing $z = W(1+\tau)$, we get that the algorithm reaches this configuration once again while the entire window consists of zeros. This is a contradiction since $\sigma_1, \sigma_2 \neq 0$, and the algorithm cannot answer both w_1 and $w_1 \cdot 0^{z(j_2-j_1)}$ correctly. \square

B Proof of Theorem 2

Before we prove Theorem 2, we start with a simpler lower bound.

Lemma 7. *Let $\epsilon < 1/4$. Any deterministic algorithm that solves the (W, τ, ϵ) -ADDITIVE SUMMING problem must use at least $\log(W/\epsilon) - O(1)$ bits.*

Proof. Denote by $\text{rep}(x) \triangleq (x \bmod R) \cdot R^{\lfloor x/R \rfloor}$ a sequence in $\{\sigma R^* \mid \sigma \in [R]\}$ whose sum is x . Next, consider the following languages:

$$L_{A_1} \triangleq \{\text{rep}(k \cdot 2RW\epsilon) \mid k \in [\lfloor 1/4\epsilon \rfloor] \setminus \{0\}\}; \quad \overline{L_{A_1}} \triangleq 0^{W+W\tau} \cdot L_{A_1} \cdot \{0^q \mid q \in [\lfloor W/2 \rfloor]\}.$$

First, notice that $|L_{A_1}| = \lfloor 1/4\epsilon \rfloor$ and that all words in L_{A_1} have length of at most $W/2$. This means that $|\overline{L_{A_1}}| = \lfloor 1/4\epsilon \rfloor \lfloor W/2 + 1 \rfloor > \lfloor W/8\epsilon \rfloor$.

We now show that every word in $\overline{L_{A_1}}$ must have a dedicated memory configuration, thereby implying a $\lceil \log \lfloor W/8\epsilon \rfloor \rceil$ bits bound. Let $w_1 = x_1 0^{q_1}$ and $w_2 = x_2 0^{q_2}$ be two distinct words in $\overline{L_{A_1}}$ such that $x_1, x_2 \in L_{A_1}$ and $q_1, q_2 \in \lfloor W/2 \rfloor$. If $x_1 \neq x_2$, then their most recent W elements differ by more than $2RW\epsilon$ and there is no output that is correct for both. Note that the slack of both w_1 and w_2 is all zeros. Hence, w_1 and w_2 require different memory configurations.

Assume that $x_1 = x_2$ and that by contradiction both w_1 and w_2 reached the same memory configuration. Since $w_1 \neq w_2$ and $x_1 = x_2$, then $q_1 \neq q_2$ and without loss of generality $q_1 < q_2$. This implies that w_1 is a prefix of w_2 so that $w_2 = w_1 \cdot 0^{q_2 - q_1}$. Thus, \mathbb{A} enters the shared configuration after reading w_1 and revisits it after reading $0^{q_2 - q_1}$. \mathbb{A} is a deterministic algorithm and therefore it reaches the same configuration also for the following word: $w_1 \cdot 0^{(W+W\tau)(q_2 - q_1)}$. In that word, the last $W+W\tau$ elements are all zeros while the sum of the last W elements in w_1 is at least $2RW\epsilon$. Hence, there is no return value that is correct for both w_1 and $w_1 \cdot 0^{(W+W\tau)(q_2 - q_1)}$. \square

We are now ready to prove Theorem 2. The theorem says that for $\epsilon < 1/4$, any deterministic algorithm \mathbb{A} that solves the (W, τ, ϵ) -ADDITIVE SUMMING problem requires $\max \{ \log(W/\epsilon) - O(1), \lceil \tau^{-1}/2 \rceil \log \lfloor \tau/2\epsilon \rfloor \}$ bits.

Proof. Lemma 7 shows that \mathbb{A} must use at least $\log(W/\epsilon) - O(1)$ bits. Given $x \in [RW\tau]$, we denote by $\text{rep}(x) \triangleq 0^{2W\tau - \lfloor x/R \rfloor - 1} \cdot (x \bmod R) \cdot R^{\lfloor x/R \rfloor}$ a sequence of the following form: $\{0^{W\tau+i}\sigma R^{W\tau-i-1} \mid i \in [W\tau - 1], \sigma \in [R]\}$ whose sum is x . We consider the following languages:

$$L_{A_2} \triangleq \{\text{rep}(k \cdot 2RW\epsilon) \mid k \in [\tau/2\epsilon]\}; \quad \overline{L_{A_2}} \triangleq \{w_1 \cdot w_2 \cdots w_{\lceil \tau^{-1}/2 \rceil} \mid \forall i : w_i \in L_{A_2}\}.$$

Our goal is to show that no two words in $\overline{L_{A_2}}$ have the same memory configuration. Let $S_1, S_2 \in L_{A_2}$ so that $S_1 \neq S_2$. Denote $S_1 \triangleq w_{1,1} \cdot w_{2,1} \cdots w_{\lceil \tau^{-1}/2 \rceil, 1}$ and $S_2 \triangleq w_{1,2} \cdot w_{2,2} \cdots w_{\lceil \tau^{-1}/2 \rceil, 2}$, while $\forall i : w_{i,1}, w_{i,2} \in L_{A_2}$. We denote $\chi \triangleq \max \{i \mid w_{i,1} \neq w_{i,2}\}$ – the last place in which S_1 differs from S_2 .

Next, consider the following sequences: $S_1^* = S_1 \cdot 0^{2W(\chi-1)\tau}$ and $S_2^* = S_2 \cdot 0^{2(\chi-1)W\tau}$. The last W elements in S_1^* are $w_{\chi,1} \cdots w_{\lceil \tau^{-1}/2 \rceil, 1} \cdot 0^{2W(\chi-1)\tau}$ and in S_2^* $w_{\chi,2} \cdots w_{\lceil \tau^{-1}/2 \rceil, 2} \cdot 0^{2W(\chi-1)\tau}$. Additionally, notice that the preceding $W\tau$ elements in both S_1^* and S_2^* are all zeros. Now, since the sum of $w_{\chi,1}, w_{\chi,2}$ must differ by at least $2RW\epsilon$, no number can approximate both with less than $RW\epsilon$ error. \square

C Correctness Proof of Algorithm 1

Theorem 10. *Algorithm 1 solves the (W, τ) -EXACT SUMMING problem.*

Proof. First, notice that c is always in the range $\{0, 1, \dots, W\tau - 1\}$ and thus the slack size is as needed. Next, assume that the algorithm input was x_1, x_2, \dots, x_t ; since stream is broken into blocks of size $W\tau$, where c is the offset within the current block, we have that $c = t \bmod W\tau$. Further, the last c elements are x_{t-c+1}, \dots, x_t and the preceding W elements are $x_{t-c-W+1}, \dots, x_{t-c}$. Finally, the algorithm always keep the sum of the W elements before the current block in B , and the sum of the current block in y ; thus, by returning $\hat{S} = B + y$ we get exactly the sum of the last $W + c$ elements. \square

D Analysis of Algorithm 3

We now analyze the memory requirement of our algorithm.

Theorem 11. *Algorithm 3 requires $O(\tau^{-1}(\log \log(RW\tau) + \log \epsilon^{-1}) + \log(RW))$ bits.*

Proof. Since $\rho \in \left(\{-\infty\} \cup \left\{0, 1, \dots, \left\lfloor \log_{(1+\epsilon/2)} RW\tau \right\rfloor\right\}\right)$, it can be represented using $\left\lceil \log \left(\log_{(1+\epsilon/2)}(RW\tau) + 2 \right) \right\rceil$ bits. Next, notice that this satisfies:

$$\begin{aligned} \left\lceil \log \left(\log_{(1+\epsilon/2)}(RW\tau) + 2 \right) \right\rceil &= \log \left(\frac{\log RW\tau}{\log(1+\epsilon/2)} \right) + O(1) \\ &= \log(\log RW\tau) - \log \log(1+\epsilon/2) + O(1) \\ &= \log(\log RW\tau) - \log \left(\frac{\ln(1+\epsilon/2)}{\ln 2} \right) + O(1) \\ &= \log(\log RW\tau) - \log \left(\frac{\epsilon/2}{\ln 2} + O(\epsilon^2) \right) + O(1) \\ &= \log(\log RW\tau) + \log \epsilon^{-1} + O(1). \end{aligned}$$

Each counter in b now is assigned a ρ value and thus the overall space consumption of b is $\tau^{-1}(\log(\log RW\tau) + \log \epsilon^{-1} + O(1))$ bits. Our B variable sums the values rather than the exact sum of each block and is a fixed point variable. We use $\lceil \log(RW\tau + 1) \rceil$ bits for its integral part and another $\lceil \log(4\epsilon^{-1}) \rceil$ for its fractional part. Thus, the total number of bits required by the algorithm is

$$\begin{aligned} \tau^{-1}(\log \log(RW\tau) + \log \epsilon^{-1} + O(1)) + O(\log(RW)) \\ = O(\tau^{-1}(\log \log(RW\tau) + \log \epsilon^{-1}) + \log(RW)). \end{aligned}$$

□

We thus get that Algorithm 3 is optimal under some conditions; notice that this includes constant τ values.

Theorem 12. *Let $\mathcal{B} = \Omega(\log(W/\epsilon) + \log \log(RW\epsilon) + \tau^{-1}(\log \epsilon^{-1} + \log \log \log(RW\tau)))$ be the (W, τ, ϵ) -MULTIPLICATIVE SUMMING lower bound showed in Theorem 3. Then for $\tau^{-1} = O\left(\frac{\log(W/\epsilon)}{\log \log W + \log \epsilon^{-1}}\right)$, $R = W^{O(1)}$, Algorithm 3 uses $O(\mathcal{B})$ bits.*

Next, we prove that Algorithm 3 solves the problem. Recall that for a real number x , we denote $(x)_{\downarrow} \triangleq \lfloor x \cdot k \rfloor / k$, for $k \triangleq \lceil \frac{4}{\epsilon} \rceil$.

Lemma 8. *Let $\epsilon \leq 1/2$; for every $y \in \mathbb{R}$ such that $y > 0$ the following inequality holds:*

$$\frac{y}{1+\epsilon} < \left((1+\epsilon/2)^{\lfloor \log_{(1+\epsilon/2)} y \rfloor} \right)_{\downarrow} \leq y$$

Proof. Observe that we have

$$\left((1+\epsilon/2)^{\lfloor \log_{(1+\epsilon/2)} y \rfloor} \right)_{\downarrow} > \left((1+\epsilon/2)^{(\log_{(1+\epsilon/2)} y) - 1} \right)_{\downarrow} = \left(\frac{y}{(1+\epsilon/2)} \right)_{\downarrow} \geq \frac{y}{(1+\epsilon/2)} - \epsilon/4.$$

Finally, since $y \geq 1$ and $\epsilon \leq 1/2$, we get that $\frac{y}{(1+\epsilon/2)} - \epsilon/4 \geq \frac{y}{1+\epsilon}$. □

Theorem 13. *Algorithm 3 solves the (W, τ, ϵ) -MULTIPLICATIVE SUMMING problem.*

Proof. First, observe that at all times $c \in [W\tau - 1]$ as needed. Denote the stream by $S = x_1, x_2, \dots, x_{t+c}$, such that c represents the number of elements within the current block. We will

show that Algorithm 3 provides a $(1 + \epsilon)$ approximation to the sum of the last $W + c$ elements $(x_{t-W+1} \dots x_{t+c})$. That is, the quantity we approximate is

$$S \triangleq \sum_{\ell=1}^{W+c} x_{t-W+\ell}.$$

Next, since we reset y in every block (Line 8), we have that at the time of the query, $y = \sum_{\ell=t+1}^{t+c} x_{\ell}$. Further, since every block is summed individually and then rounded in Line 5. We assume without loss of generality that $i = 0$ at the time of the query, and $\forall j \in [\tau^{-1} - 1]$ we denote $S_j \triangleq \sum_{\ell=t-W\tau(\tau^{-1}-j)+1}^{t-W\tau(\tau^{-1}-j-1)} x_{\ell}$ – the sum of the elements in block j . Therefore we have:

$$\forall j \in [\tau^{-1} - 1] : b_j = \left\lfloor \log_{(1+\epsilon/2)} S_j \right\rfloor. \quad (4)$$

That is each of the τ^{-1} blocks that precede the last c elements is summed exactly (Line 2) and then we store its $(1 + \epsilon/2)$ -based log in b . Next, the B variable stores the approximated values rounded down (Line 6), i.e.,

$$B = \sum_{j=0}^{\tau^{-1}-1} ((1 + \epsilon/2)^{b_j})_{\downarrow}. \quad (5)$$

Now, Lemma 8 implies that

$$\forall j \in [\tau^{-1} - 1] : S_j > 0 \implies \frac{S_j}{1 + \epsilon} < \left((1 + \epsilon/2)^{\lfloor \log_{(1+\epsilon/2)} S_j \rfloor} \right)_{\downarrow} \leq S_j. \quad (6)$$

We then get

$$S = \sum_{\ell=1}^{W+c} x_{t-W+\ell} = \sum_{\ell=1}^W x_{t-W+\ell} + \sum_{\ell=W+1}^{W+c} x_{t-W+\ell} = \sum_{j=0}^{\tau^{-1}-1} S_j + y = \sum_{\substack{j \in [\tau^{-1}-1]: \\ S_j > 0}}^{\tau^{-1}-1} S_j + y. \quad (7)$$

Also, note that if some $S_j = 0$, then we defined b_j as $-\infty$ and $(1 + \epsilon/2)^{b_j}$ as 0. This means that if $S = 0$, then $\forall j \in [\tau^{-1} - 1] : S_j = 0, y = 0$ and thus $\hat{S} = 0$. Hereafter, assume that $S \neq 0$. Next, we plug equations (4),(5),(6) into (7) to get:

$$S = \sum_{\substack{j \in [\tau^{-1}-1]: \\ S_j > 0}}^{\tau^{-1}-1} S_j + y \geq \sum_{\substack{j \in [\tau^{-1}-1]: \\ S_j > 0}}^{\tau^{-1}-1} ((1 + \epsilon/2)^{b_j})_{\downarrow} + y = B + y = \hat{S}.$$

Similarly, we bound \hat{S} from below as follows:

$$\frac{S}{1 + \epsilon} = \frac{\sum_{\substack{j \in [\tau^{-1}-1]: \\ S_j > 0}}^{\tau^{-1}-1} S_j + y}{1 + \epsilon} < \sum_{\substack{j \in [\tau^{-1}-1]: \\ S_j > 0}}^{\tau^{-1}-1} ((1 + \epsilon/2)^{b_j})_{\downarrow} + y = B + y = \hat{S}.$$

We showed that in all cases where $S \neq 0$ we have $\frac{S}{1+\epsilon} < \hat{S} \leq S$ if $S > 0$, thereby proving the theorem. \square

E Hyper LogLog (HLL) Overview

We provide a brief overview of the HLL algorithm [22]. HLL uses a hash function that maps each identifier to an infinite string of 0,1 bits; $H : \mathbb{D} \rightarrow \{0,1\}^\infty$, where \mathbb{D} is the identifier domain. Given $s \in \{0,1\}^\infty$, the operator $lsb(s)$ returns the position of the leftmost 1-bit, e.g., $lsb(0001\dots) = 3$ (counting from 0). Intuitively, a string with k leading zeros is expected to appear after 2^k events. That is, *HLL* stores the largest previously encountered lsb value in a counter. The space complexity is $\log \log (|\mathcal{M}|)$, which in practice means that a single byte suffices for most scenarios.

To augment precision, m different *HLL* counters are used. For simplicity, we assume that $m = 2^b$ for some positive $b \in \mathbb{N}$. Stochastic averaging is performed by using the first b bits of each hashed value to determine the *HLL* instance to be updated.

To satisfy a query, we read all m estimations, calculate their harmonic average and normalize the result. The technical details of how to best interpret the result can be found in [22, 25]. Algorithm 5 provides pseudo code of the HLL algorithm. As can be observed, stochastic averaging is performed in Line 5. The notation $(h_0, h_1, \dots, h_{b-1})$ describes the bit composition of h . The query algorithm uses the harmonic average of the different experiments $(M[0], M[1], \dots, M[m])$ and the result is then normalized with a constant that depends on m (specifically, $\alpha_m \cdot m^2$).

Algorithm 5 HyperLogLog (for streams)

```

1: Initialization:  $M[0] = M[1] = \dots = M[m-1] = -\infty$ .
2: function UPDATE(ELEMENT  $x$ )
3:    $h = H(x)$ 
4:    $id = (h_0, h_1, \dots, h_{b-1})$  ▷ We use the first  $b = \log m$  bits to determine which register to update
5:    $w = (h_b, h_{b+1}, \dots)$  ▷ We compute the number of leading zeros  $\rho(w)$  in the following bits.
6:    $M[id] = \max(M[id], \rho(w))$ 
7: function QUERY(ELEMENT  $x$ )
8:    $Z = \left( \sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$ 
9:   return  $\alpha_m \cdot m^2 \cdot Z$ .
```

F Proof of Theorem 9

Before we prove Theorem 9, we need to prove an auxiliary lemma.

Lemma 9. *The last $\min(|\mathcal{M}|, W + PB)$ updates are performed on all HLL instances.*

Proof. Initially, when $|\mathcal{M}| < W + W\tau$, every *HLL* instance that is initialized is already empty. Thus, since every Slacky HLL updates an *HLL* instance (in Line 2), the number of update operations is $|\mathcal{M}|$. When $|\mathcal{M}| = W + W\tau$, *HLL*[0] is initialized at first and we lose the oldest τW events. At that point, the number of events is W and $PB = 0$. In each subsequent update, both PB and the number of events are increased by 1 until $PB = \tau W$ again. At that point, PB is set back to 0 and the number of events drops to the last W . \square

We are now ready to prove Theorem 9.

Proof. Note that $W' = W + PB$ and therefore Lemma 9 guarantees that the last W' events are summarized in Slacky HLL. Therefore, $M_{max}[i]$ used to generate Z in Slacky HLL has the highest ρ value of all the last W' elements. Consequently, the value that determined $M_{max}[i]$ in Slacky HLL also determines $M[i]$ in an HLL that summarizes the last W elements. \square